SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 199–224 (1996)

Practice

Software Metrics and Microcode: A Case Study

GEORGE TRIANTAFYLLOS
IBM Corporation, LSCD Division, Poughkeepsie, NY 12601, U.S.A.

STAMATIS VASSILIADIS
Department of Electrical Engineering, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

JOSÉ G. DELGADO-FRIAS
Department of Electrical Engineering, Binghamton University, Box 6000, Binghamton, NY 13902, U.S.A.

SUMMARY

In this paper, we report the findings of an investigation undertaken at IBM[1] to determine whether or not existing software metrics are applicable to the microcode of large computer systems. As part of this investigation, we calculated several metrics from the microcode developed for the IBM 4381 and IBM 9370 computer systems, and used them as predictive parameters for a number of existing error prediction models. The microcode used in this case study exceeds 1·2 million lines of code written in 12 languages and comprises the microcode for the IBM ES/4381[2] and IBM ES/9370[3] computer systems. Our results suggest that only a few of the existing metrics are linearly independent, and that none of the metrics examined can be used in a regression model as a reliable error predictor.

KEY WORDS: software metrics; microcode metrics; error prediction models; static models; regression models; code metrics

1. INTRODUCTION

The numerous quantitative measures of program 'quality', collectively referred to as software metrics (Kafura and Canning, 1985), may be divided into three categories, namely:

1. code metrics: such as lines of code, program effort (Halstead, 1977), and cyclomatic complexity (McCabe, 1976);
2. structure metrics: such as the information flow metric (Henry, 1979), the invocation complexity metric (McClure, 1978), and the review complexity measure (Woodfield, 1980); and

[1] IBM is a registered trademark of the International Business Machines Corporation.
[2,3] S/4381 and ES/9370 are trademarks of the International Business Machines Corporation.

CCC 1040–550X/96/030199–26

Received 25 May 1994
Revised 12 November 1995

3. hybrid metrics: which are modifications of the structure metrics weighted by some code metric.

Using the above metrics, a number of models have been proposed to estimate the number of errors incurred during the development of a software product and/or the quality, and reliability of the final product. Examples of such models may be found in Conte, Dunsmore and Shen (1986).

While software products have been studied extensively in the past, the microcode of computer systems has been neglected. In large computer systems, the microcode comprises a large portion of the development and maintenance cycles, consisting of hundreds of thousands of lines of code, written in assembly and/or high level languages. Like 'regular' software, microcode undergoes the same design and coding techniques, and thus, it can be viewed as a software product. Given the size and complexity of the microcode, performing a verification of its function is an expensive and time consuming process. In fact, the functional testing of computer systems, and in particular, the functional testing of its microcode, constitutes a large portion, often the largest, of the development and maintenance cycles. Given the cost associated with the functional testing and the demand for the shortening of the development and maintenance cycles, anticipating the 'deviations' from the pre-specified requirements to be found during the functional testing phase constitutes one of the most interesting problems in computer development.

As indicated earlier, the microcode resembles software products, thus it is necessary to establish if the software metrics and models developed for software systems have any validity to the microcode development. In this paper we discuss our findings regarding the software metrics and their applicability to microcode. In particular, we consider:

1. microcode metrics relationships, to determine if existing software science metrics are applicable to the microcode, and
2. we investigate the applicability of regression models in the microcode.

The relationship among various metrics is investigated using linear regression between pairs of metrics. A second issue examined here is the issue of the best metric among several metrics that measure the same microcode attribute. Third, the possibility of using metrics to estimate the errors to be found during the testing phases is examined. In particular, we use one or more metrics in a regression model to determine whether the metrics can predict the errors found in the microcode.

This study is of particular importance for three reasons: first, it is the first study to be published that examines the applicability of software science metrics to microcode. Second, it is one of the first studies to use such high volume of data. Similar studies in software systems have used considerably less amounts of data. From the studies we surveyed (Smith, 1980; Basili, Selby and Phillips, 1983; Motley and Brooks, 1977; Paulsen, Fitsos and Shen, 1983; Shen *et al.*, 1985) we were able to find only one that was conducted on a single product having about half a million lines of code (Paulsen, Fitsos and Shen, 1983). Third, as far as we know, this is the first study to include the defects found in the entire life cycle. Most studies we surveyed include defects found only in the later phases of the development cycle.

The presentation is organized as follows: the data used in this study are presented in the next section. Subsequently, the metrics used in the microcode are presented and their

relationships are investigated. Following that we examine software science as it applies to microcode. Finally we examine the possibility of using the metrics in linear and multilinear error prediction models to estimate the errors to be found in the microcode.

2. DATA COLLECTION

In establishing the applicability and the predictive capabilities of software metrics to microcode we consider, as a case study, the microcode developed for the IBM 4381 and the IBM 9370 computer systems. Figure 1 shows a summary of the source code for these two systems. Each system was divided into three units, namely the processor unit (PU) the input/output unit (IO) and the support processor unit (SP). Each unit was further divided into sub-units, one for each programming language used in the development, resulting in 14 groups of modules. For each sub-unit, Figure 1 shows the language the sub-unit was written in, the number of modules that comprised the sub-unit, the deliverable lines of code (DLOC), the lines-of-code (LOC), and the defects found in each sub-unit. In Figure 1, languages denoted as ASM1 through ASM5 refer to assembler languages used internally at IBM. The code of the two systems combined totals 4635 modules, over 2 million DLOC, 1·2 million LOC, and was developed using 12 distinct languages. Almost half of the code was written in high level languages.

The metrics measured include Halstead's software science metrics, such as the unique and total operators and operands, denoted as n_1, N_1, n_2, N_2; the observed and estimated program length, N, $\hat{N}$; the program volume, V; the program level and difficulty, L, D; the program complexity C; etc..

Unit	Language	Modules	DLOC	LOC	defects
IBM 4381 Computer System					
PU	ASM1	347	173,535	104,984	4,165
IO	ASM1	112	92,181	35,080	2,565
SP	ASM2	626	316,893	210,459	6,016
	SPIEL	211	63,667	42,741	
	ASM3	209	145,420	87,428	
totals	4 languages	1505	791,696	480,692	12,746
IBM 9370 Computer System					
PU	ASM4	309	214,075	72,834	3,069
IO	ASM5	154	49,858	21,680	4,043
	PL8	1,096	255,993	119,734	
	680ASM	111	16,038	5,942	
SP	ASM4	148	204,645	108,025	10,580
	370 ASM	56	20,385	12,820	
	MASM	75	15,762	10,444	
	ASM86	46	40,305	35,552	
	PLS86	1,135	695,689	409,105	
totals	8 languages	3,130	1,512,750	796,136	17,692

Figure 1. Source code of the IBM 4381 and the IBM 9370 microcode divided into units and sub-units

Figures 2 and 3 show some representative statistics of the preliminary analysis of the code. Readers are referred to Triantafyllos, Vassiliadis and Delgado-Frias (1993b) for more information on the entire data set. The modules are grouped together in every ten LOC. The great majority of the modules have between 0 and 400 LOC. Practically, there are no modules with more than 2000 LOC. Similar concentrations have been observed in all sub-units. The modules were classified in five groups depending on the module size. This classification was done according to Figure 4. Figure 5 shows the percentage of modules of the two systems that fall in each category. As the figure suggests, most modules are small or medium (72·84%). Less than 8% of the modules have less than 10 LOC. About 20% of the modules fall into the large and very large category.

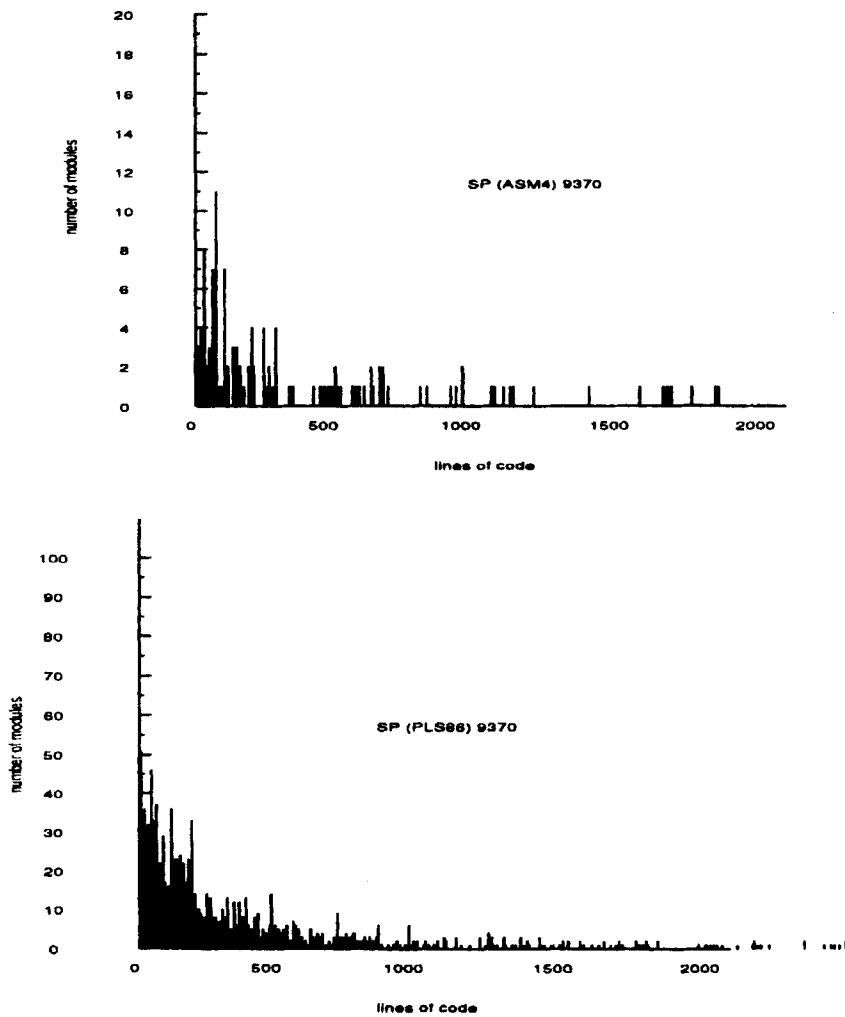


Figure 2. Number of programs in the 9370 PU and IO units as function of LOC

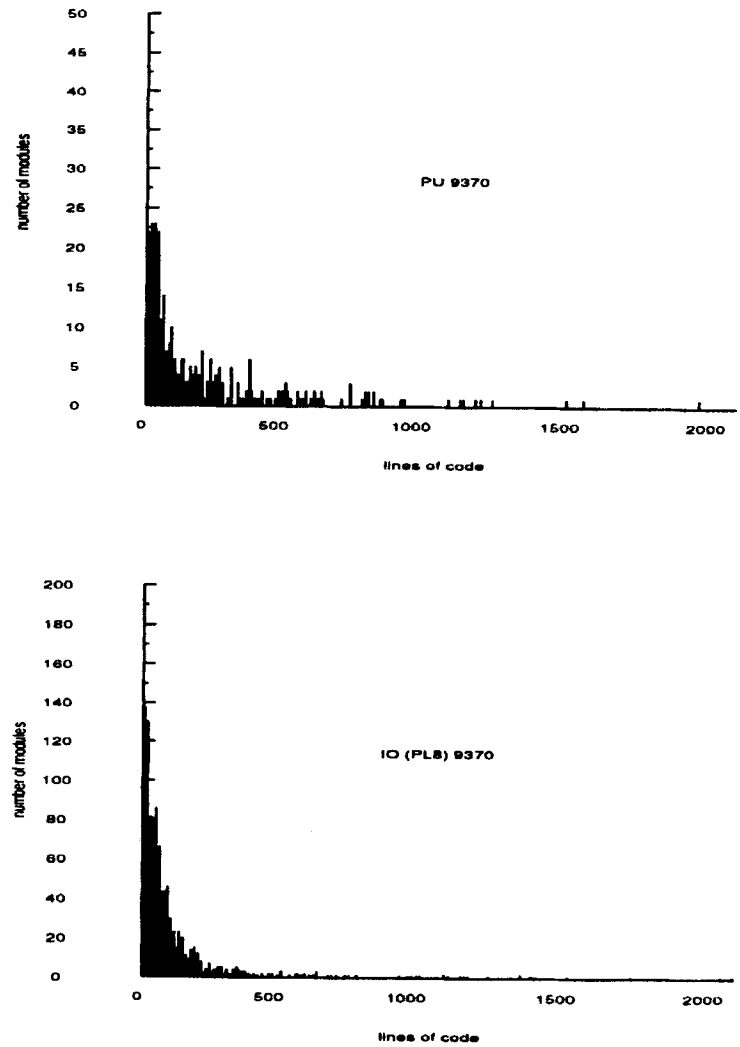


Figure 3. Number of programs in the 9370 SP unit as function of LOC

class	LOC	N
noise	< 10	< 100
small	10 - 99	100 - 2000
medium	100 - 399	2000 - 4000
large	400 - 999	> 4000
very large	> 999	

Figure 4. Module classification by lines of code (LOC) and program length (N)

	LOC				N			
class	modules	%	av. LOC	st. dev	modules	%	ave. N	st. dev.
noise	355	7.67	6.35	2.13	843	18.19	46.12	25.18
small	1787	38.55	47.59	25.89	3081	66.47	657.47	497.80
medium	1590	34.30	210.27	84.87	435	9.38	2774.45	540.45
large	625	13.48	604.67	155.52	276	5.96	7318.12	3818.87
v. large	278	6.00	1695.40	843.21	—	—	—	—

Figure 5. Percentage of modules in each module size category

3. AN ANALYSIS OF THE MICROCODE METRICS RELATIONSHIPS

In this section we discuss the relationship between several metrics of the microcode. Such a relationship is derived using the following model: metric$B = b_0 + b_1 \times$ metric$A$. We note here that in our investigations we use the general form of the model, and expect an estimate of the coefficient b_0 to be close to zero. A non-zero b_0 would then suggest an anomaly in the data or perhaps the inadequacy of a linear model. The correlation coefficient, ρ, and the coefficient of determination, R^2 are also used. (Note that ρ and R^2 are not affected from the choice of the model, thus all conclusions regarding the linearity between metrics remain valid regardless of the choice of the model.) The correlation coefficient may be loosely used to indicate how well the two variables are correlated, or to express the degree to which they depend on each other. A high value of ρ (close to 1·0) indicates that the dependant variable Y is highly dependent on X. The coefficient of determination R^2 may be interpreted as the percentage of the variation of Y that may be attributed to X. For example, if $R^2 = 0{\cdot}95$ then it may be argued that 95% of the variation in Y is associated (but not necessarily caused by) variation in X. The adequacy of the fitted model was tested by using the null hypothesis and the residuals of the regression. Details of this analysis may be found in Triantafyllos, Vassiliadis and Delgado-Frias (1993a).

In this investigation we examine if there is a linear relationship between lines of code and volume, complexity operators, operands, program length, program effort, function calls, difficulty and DLOC, by applying linear regression on each sub-unit of the two systems. We also examine the relationship between all pairs of metrics by applying the same technique on all modules of the two systems. The results of this first analysis are summarized in Figure 6, which shows the statistical correlation between LOC and each pair of metrics. As the figure suggests, there is a very strong correlation (above 0·9) between lines of code and program volume, total operators, total operands and program length. This high correlation between these variables suggests that most of the variation of each metric is associated with the variation in the lines of code. In other words, the LOC is linearly dependent on program volume, operands, operators and length. This finding is very significant in metrics experimentations using regression models because, to a large extent, the lines of code metric is able to capture most of the information. This finding suggests that in regression models there is no significant advantage to using volume, length, operands, operators or linear combinations of them instead of LOC. Since the correlation between LOC and complexity, effort and function calls is not as strong as

unit	Lang.	volume	complex.	operators	operands	length	effort	f. calls	diff.	DLOC
IBM 4381 Computer System										
PU	ASM1	0.916	0.722	0.935	0.923	0.929	0.850	0.406	0.804	0.945
IO	ASM1	0.990	0.935	0.986	0.991	0.989	0.915	0.390	0.869	0.973
	ASM2	0.973	0.796	0.974	0.969	0.973	0.867	0.814	0.759	0.964
SP	SPIEL	0.983	0.855	0.956	0.951	0.970	0.951	0.856	0.707	0.982
	ASM3	0.980	0.845	0.989	0.983	0.987	0.895	0.885	0.816	0.927
IBM 9370 Computer System										
PU	ASM4	0.988	0.816	0.987	0.990	0.992	0.833	0.610	0.883	0.945
IO	ASM5	0.942	0.754	0.950	0.961	0.960	0.853	0.878	0.758	0.932
	PL8	0.968	0.858	0.976	0.941	0.970	0.893	0.136	0.857	0.967
	680ASM	0.991	0.975	0.994	0.994	0.994	0.946	0.873	0.753	0.991
SP	ASM4	0.995	0.732	0.993	0.992	0.996	0.957	0.863	0.885	0.974
	370 ASM	0.970	0.871	0.976	0.968	0.972	0.921	0.916	0.882	0.993
	MASM	0.998	0.990	0.995	0.996	0.996	0.988	0.965	0.951	0.997
	ASM86	0.991	0.001	0.999	0.985	0.993	—	-0.007	-0.121	0.941
	PLS86	0.981	0.934	0.981	0.968	0.978	0.827	0.783	0.914	0.972

Figure 6. Correlation of lines of code with the complexity, operands and operators metrics

the correlation between LOC and the other parameters, it may be argued that the program complexity, effort or function calls may, in a number of cases, provide additional information.

Figure 7 provides a representative plot of the scatter diagrams and the fitted model of each sub-unit. Each mark on the scatter diagram represents a module. The value on the *X* axis represents the LOC for that module, and the value on the *Y* axis represents the value of each corresponding metric for a given module. The data are plotted in linear

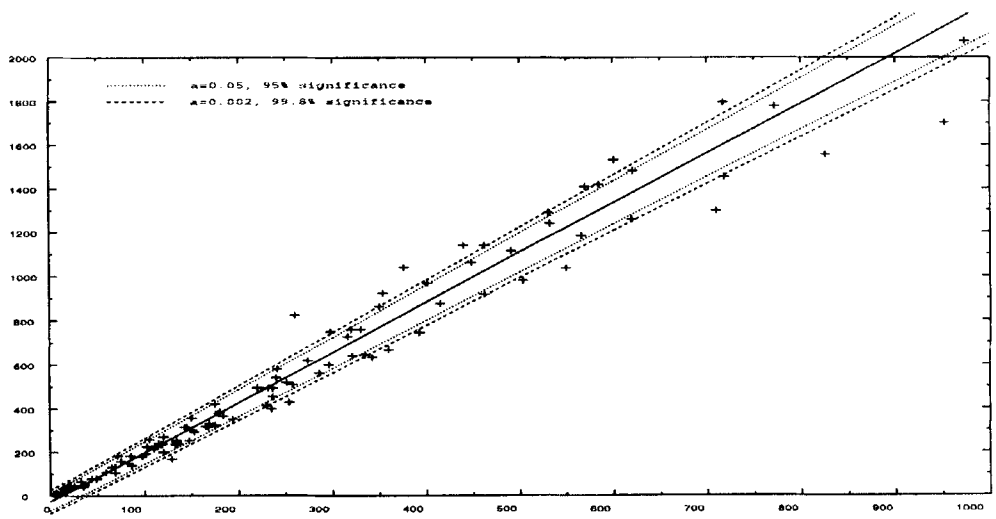


Figure 7. IO 4381: LOC versus total operators

scale. In many cases, a number of modules have exceptionally large metrics. These modules are taken into account for the numerical results but they are not included in the graphs because their inclusion would result in significant compression of the graphs.

All metrics examined, except the programming effort, showed a strong or weak (as in the case of complexity) linear relationship with LOC. The program effort did not correlate with the LOC. Figure 8 shows the scatter diagram of LOC and the estimated program effort $\bar{E}$ (Halstead, 1977) of the PLS/86 sub-unit of the 9370 system. As the figure suggests, the programming effort increases exponentially with the size of the program. The quadratic model $E = b_0 + b_1\text{LOC} + b_2\text{LOC}^2$ was used to fit the data. It has been suggested that effort relates to program size according to $E \approx aN^b + c$ (power model) where b was found to range from $b \approx 0{\cdot}91$ to $b \approx 1{\cdot}83$ for large modules (Walston and Felix, 1977). Both the quadratic and the power model gave comparable results.

The fact that LOC is strongly related to volume and length can also be seen in Figure 9 which shows the LOC, length and volume metric of all 4635 modules plotted against the module index and sorted by LOC. Note that the values of the volume metric were scaled by 0·01 in order to fit the graph. As the figure suggests, within certain limits all three metrics may be considered as one curve, scaled appropriately. In contrast, Figure 10 shows the same information for two 'less correlated' metrics, the LOC, scaled by 0·5, and the complexity.Unlike Figure 9, in this case the two curves cannot be considered as the same, scaled appropriately, regardless of the bandwidth allowed. Plots similar to Figure 10 were obtained for all non-related metrics. Figure 10 shows that there are many modules with small complexity whose size (in LOC) varies from close to zero lines to 2000 lines (keep in mind that LOC is scaled by 0·5 in the figure, thus a module that shows 1000 lines has in reality 2000 lines). This observation may be explained by 'certain programming practices'. For example, it is customary in programming to place items such as data tables and/or declarations, in a single module or modules and keep the code that manipulates the data in different modules. This may explain why there are modules with 5000 LOC and practically zero complexity.

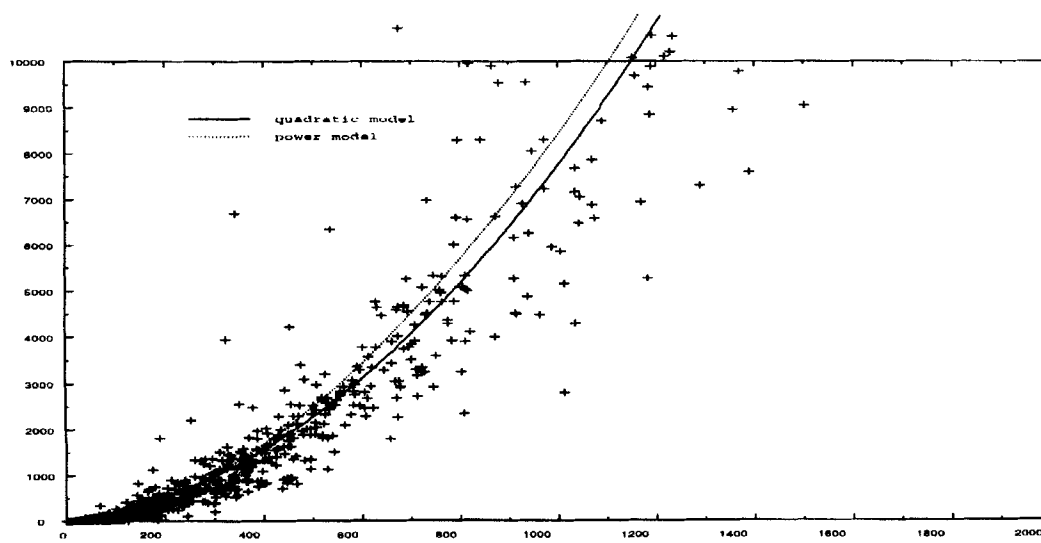


Figure 8. SP 9370 (PLS/86): LOC versus program effort. (Effort in thousands)

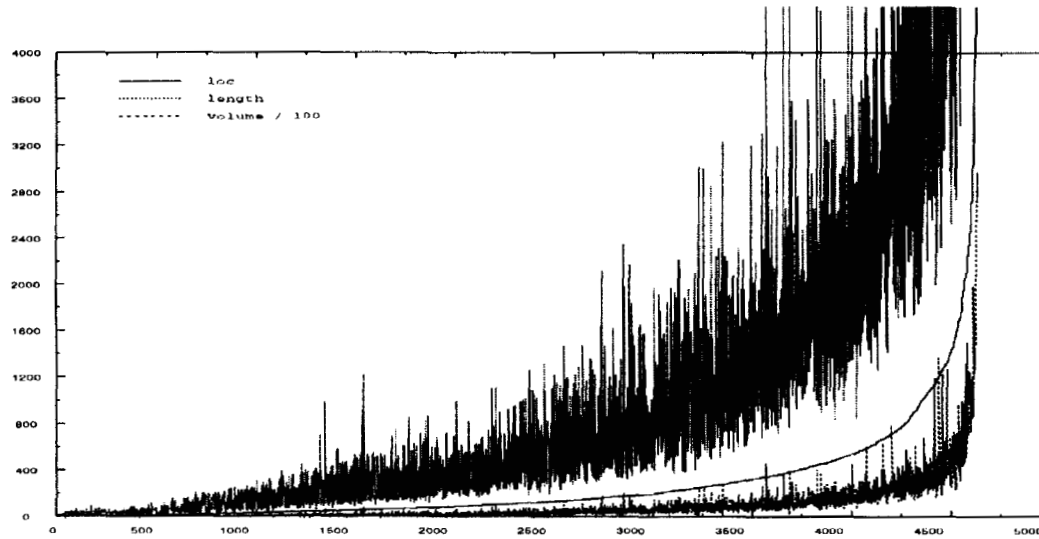


Figure 9. Module index versus LOC, length and volume sorted by LOC of all 4635 modules

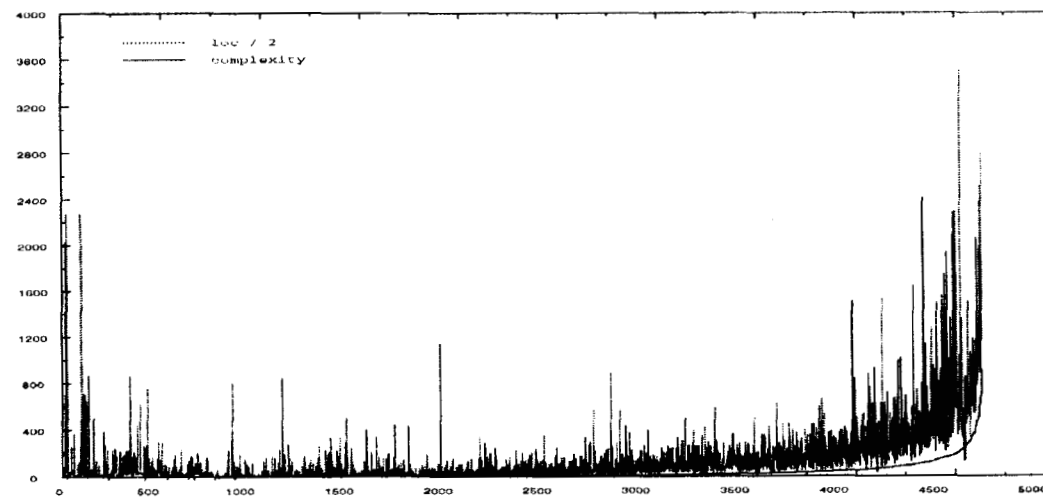


Figure 10. Module index versus LOC and complexity sorted by LOC of all 4635 modules

Figure 11 shows the correlation between all metrics examined computed by putting together all 4635 modules of the microcode we examined. The figure shows how each metric is correlated to another. However, a comparison of this figure with the figures that show the correlation of LOC with each metric shows that grouping all modules together in one 'system' may not be appropriate. For example, Figure 6 shows that the correlation of LOC with total operators is well above 0·93 for each of the 14 sub-units. When grouping all modules together the correlation of the two metrics drops to 0·881, as shown in Figure 11. The reason for the lower correlation is that even though LOC and operators are strongly correlated, the regression line has a different slope for each sub-unit, thus,

	LOC	DLOC	volume	complex.	operands	operators	length	effort	f. calls
LOC	1.0	0.928	0.949	0.749	0.943	0.881	0.949	0.768	0.610
DLOC	0.928	1.0	0.861	0.672	0.865	0.798	0.857	0.738	0.537
volume	0.949	0.861	1.0	0.776	0.970	0.958	0.994	0.828	0.670
complexity	0.749	0.672	0.776	1.0	0.688	0.844	0.789	0.648	0.616
operands	0.943	0.865	0.970	0.688	1.0	0.879	0.969	0.753	0.553
operators	0.881	0.798	0.958	0.844	0.879	1.0	0.969	0.822	0.774
length	0.949	0.857	0.994	0.787	0.969	0.969	1.0	0.812	0.683
effort	0.768	0.738	0.828	0.648	0.753	0.822	0.812	1.0	0.657
f. calls	0.610	0.537	0.670	0.616	0.553	0.774	0.683	0.657	1.0

Figure 11. Correlation between metrics for all 4635 modules

when all modules are grouped together, they result in a 'spread' scatter diagram. Figure 12 shows the patterns of the multiple regression lines resulting from grouping together all 4635 modules.

4. AN EVALUATION OF THE SOFTWARE SCIENCE IN THE MICROCODE

4.1. Software science

Software science (Halstead, 1977), an attempt for explaining the programming development process, has been extensively studied and critiqued in the past two decades. (Shen, Conte and Dunsmore, 1983) for software products. Driven by the necessity to manage the ever growing cost of software, researchers have devoted much attention to the ideas first presented by Halstead (1977), expecting to produce models and processes that explain

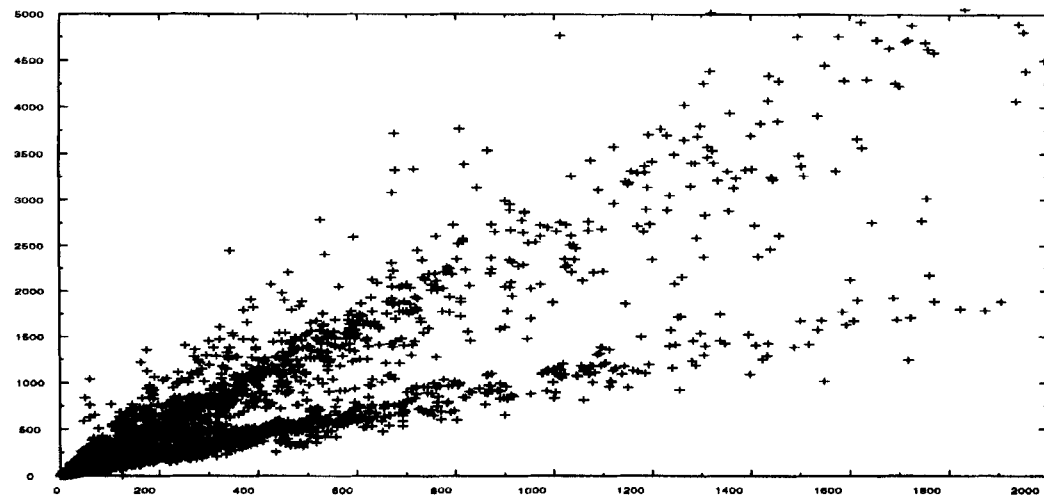


Figure 12. Lines of code versus total operators for all 4635 modules

and control the software development process. The estimation of the programming effort, the program defects and the programming cost are some of the issues dealt with for several years. Despite the increased efforts, no universally accepted models have been proposed. Simple models, based on metrics, have been shown to be applicable in certain projects but a poor fit in others. Models that include many factors also follow the same trend (Shen, Conte and Dunsmore, 1983).

In this section, we investigate certain trends in microcode programming such as the relative size of the microcode modules, the use of operands and operators, and their relationship to program size. Moreover, we report the results of our investigation concerning the consistency of software science as it applies to the program length, and the usefulness of the program difficulty in the prediction of the errors incurred during the development of the code.

4.2. Program length

According to Halstead (1977), a program is comprised of operators and operands, denoted as n_1, n_2 respectively. Their sum $n = n_1 + n_2$ is called the vocabulary of the program. The program length, denoted by N, is defined as the sum of the total operand and operator usage in a program, $N = N_1 + N_2$, and has been proposed as an alternative measure of program size (Halstead, 1977). According to the first software science hypothesis 'the length of a well-structured program is a function only of the number of unique operators and operands'. Halstead proposed that the length can be estimated from the vocabulary of the program, if the latter is known, using the following equation:

$$\overline{N} = n_1 \log_2(n_1) + n_2 \log_2(n_2) \tag{1}$$

The estimated length is useful to make early predictions about the size of a program before the code is written, assuming that the operands and operators needed to code a program can be reasonably estimated. On a system using top-level design it is possible to measure n_2^* before the coding begins and to reasonably estimate n_2 using a linear relation $n_2 = an_2^*$ (Gaffney, 1981), where the * denotes externally recognized operands or operators.

We have conducted the following experiment in order to determine the validity of the above hypothesis in the microcode. We measured the program length, computed the estimated program length according to the above equation, and measured the correlation between the two variables N, $\overline{N}$. Linear regression was used to determine the relationship between N and $\overline{N}$. In this case, we have assumed that $\overline{N}$ may be estimated from N using the equation $\overline{N} = b_0 + b_1 N$. The results, Figure 13, show correlations between the two variables and the values of the coefficients b_0,b_1 in each of the 14 sub-units. As the figure suggests many sub-units have a strong correlation between the two variables. However, in a number of cases, the program length does not correlate well with the estimated counterpart (e.g. PU 4881, PU 9370, IO 9370). The coefficients of the regression analysis fluctuate considerably from the expected values ($b_0 \equiv 0$ and $b_1 \equiv 1{\cdot}0$), causing the length equation to over or under predict N. Additional work showed that in most units the estimated length was above the actual length.

Although the accuracy on each individual module may vary considerably, the accuracy on the sub-unit level was expected to be much better because when individual modules

unit	Language	correl	b_0	b_1	unit	Language	correl	b_0	b_1
IBM 4381 Computer System					IBM 9370 Computer System				
PU	ASM1	0.776	397.047	0.885	PU	ASM4	0.776	397.047	0.885
IO	ASM1	0.935	395.269	0.567	IO	ASM5	0.892	143.279	1.378
SP	ASM2	0.964	64.723	1.640		PL8	0.889	208.322	0.741
	SPIEL	0.976	158.356	1.511		680ASM	0.941	65.912	1.552
	ASM3	0.914	318.292	1.107	SP	ASM4	0.973	397.278	0.726
						370 ASM	0.959	175.842	1.605
						MASM	0.995	-35.089	1.584
						ASM86	0.987	-1642.5	3.881
						PLS86	0.914	335.986	0.554

Figure 13. Correlation between the estimated and the actual program length

Unit	Language	Modules	LOC	Volume	N	$\overline{N}$
4381 Computer System						
PU	ASM1	347	104,984	5,395,856.0	653,491.0	468,317.7
IO	ASM1	112	35,080	1,459,396.1	189,899.0	147,463.5
SP	ASM2	626	210,459	5,768,354.2	686,087.0	1,165,690.1
	SPIEL	211	42,741	1,045,397.8	135,250.0	273,732.1
	ASM3	209	87,428	2,080,400.5	241,366.0	333,736.9
9370 Computer System						
PU	ASM4	309	72,834	1,487,171.6	185,473.0	286,906.0
IO	ASM5	154	21,680	434,523.8	54,675.0	97,430.4
	PL8	1,096	119,734	4,321,225.4	584,069.0	661,224.6
	680ASM	111	5,942	163,828.1	20,100.0	38,517.7
SP	ASM4	148	108,025	3,512,748.6	378,972.0	333,864.5
	370 ASM	56	12,820	247,763.0	30,833.0	59,343.6
	MASM	75	10,444	239,814.1	31,092.0	46,607.7
	ASM86	46	35,552	1,076,372.7	102,777.0	323,301.8
	PLS86	1,135	409,105	16,733,913.2	2,005,039.0	1,491,964.1

Figure 14. Total size of the code by LOC, volume, observed length and estimated length

are grouped together, several parameters tend to average out. This did not occur, as the accuracy of $\overline{N}$ to predict N ranges from 71·6% to 314.57% (Triantafyllos, Vassiliadis and Delgado-Frias, 1993b). Examples of the previous findings can be seen in the following figures which show the plots of the observed and estimated length of the modules of all sub-units. More information is provided in Triantafyllos, Vassiliadis and Delgado-Frias (1993b). The modules are sorted by observed length in ascending order. Figure 15 shows that the length function appears to overestimate the length for modules with actual length approximately below 1 000, and under estimate larger modules. This is in compliance with observations reported in Christensen, Fitsos and Smith (1981); Smith (1980); and Basili, Selby and Phillips (1983), but it does not occur in all sub-units. In Figure 16 large modules are under estimated but no conclusion can be made for smaller modules. In most

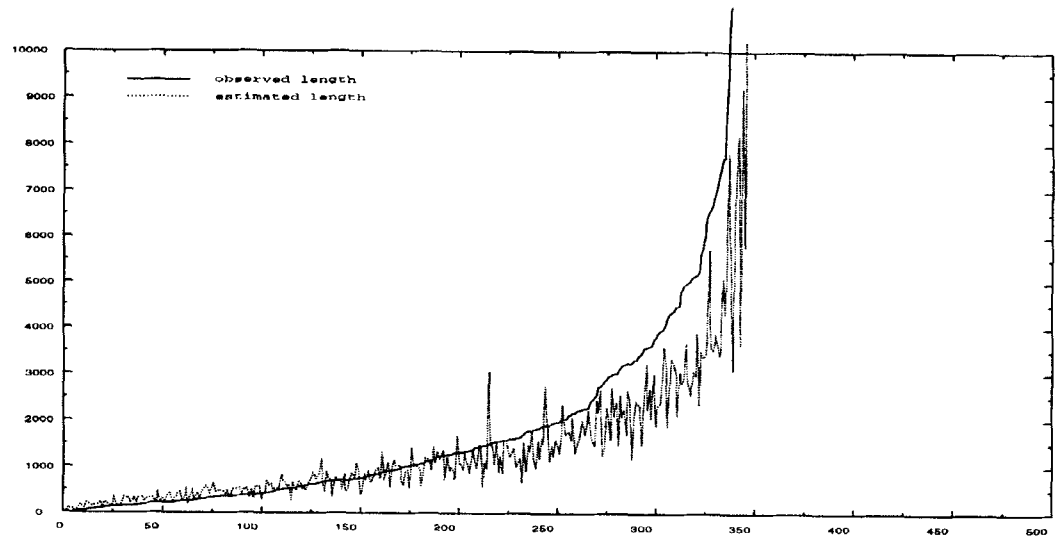


Figure 15. PU 4381: observed and estimated length plotted against module index (horizontal axis)

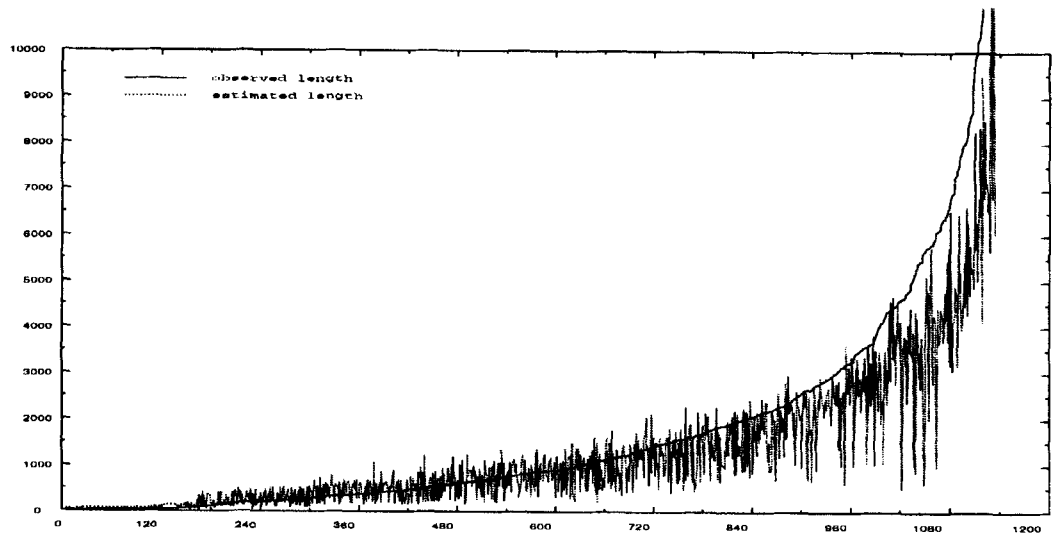


Figure 16. SP 9370 (PLS86): observed and estimated length plotted against module index (horizontal axis)

other cases, a representative example is shown in Figure 17, the length function always overestimates the actual length for both assembler and high level languages. This case study provides no conclusive evidence as to whether the length equation can compute the size of a program with reasonable accuracy.

4.3. Program difficulty

The program difficulty is defined in software science as $D = \frac{(n_1 N_2)}{2n_2})$ expressing the difficulty in writing programs. This definition of difficulty is based on a rather intuitive

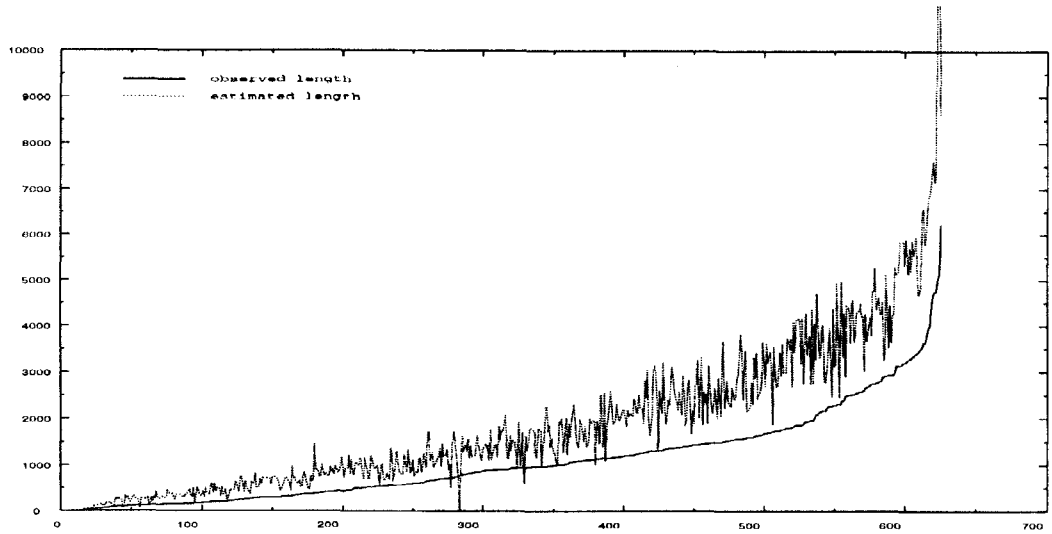


Figure 17. SP 4381 (ASM2): observed and estimated length plotted against module index (horizontal axis)

justification that the more symbols a program uses the more difficult it is. It has been argued that the term $n_1/2$ tends to remain relatively constant, and thus the average usage of operands N_2/n_2 is the main contributor to the program difficulty (Christensen, Fitsos and Smith, 1981). According to this definition, a program where each operand is used only once would have a low difficulty ($N_2/n_2 \approx 1$), which implies that an 'easy' program is one where each variable is used only once. However, this argument is as convincing as the opposite argument; that a program with one variable that changes many times is easier because one has to keep track of just one variable, instead of many.

We measure microcode program difficulty by the number of errors found in a module. Figure 18 shows some statistics of D measured in our data. For each group of modules we show the minimum and maximum difficulty, the mean, the median difficulty and the standard deviation. Comparing the difficulty ranges obtained in Figure 18 with the ones reported in Smith (1980), it seems that the difficulty of the microcode modules codes in PLS ($D = 76{\cdot}43$) is comparable with the difficulty of PLS code reported in Smith (1980), which ranges from 38 to 151. The difficulty of the assembler code in our data is considerably less than the difficulty of 370 assembler which ranges from 91 to 561. Based on these figures alone we can conclude that the assembler microcode is less difficult than 'regular' assembler code.

Figures 19 and 20 show a representative analysis taken from Triantafyllos, Vassiliadis and Delgado-Frias (1993b). In these figures a plot of the difficulty of each module sorted in ascending order and the corresponding errors is presented. There are some patterns suggesting that increased difficulty implies more errors. However, this observation does not hold true in general, since there are several modules with a constant number of errors and various difficulties as well as modules with high difficulty and less errors. The IO of the 4381 shows a more consistent increase in errors as a function of difficulty (Triantafyllos, Vassiliadis and Delgado-Frias, 1993b). The plots suggest that there is no observable relation between errors and difficulty. In general, we found that there are very few modules that have a linear relationship with difficulty.

unit	Language	low	high	mean	median	st. dev.
4381 Computer System						
PU	ASM1	3.0	511	87.15	73.67	68.67
IO	ASM1	2.8	265	81.21	73.18	55.74
SP	ASM2	1.0	385	67.60	64.65	46.32
	SPIEL	5.0	161	44.51	39.75	25.46
	ASM3	2.0	351	56.03	28.35	66.90
9370 Computer System						
PU	ASM4	1.0	709	68.18	41.82	87.59
IO	ASM5	1.5	187	24.47	14.91	28.56
	PL8	2.0	231	24.37	16.11	27.27
	680ASM	1.5	96	12.15	7.42	16.57
SP	ASM4	1.0	607	139.81	89.91	143.30
	370 ASM	0.6	147	34.33	23.63	38.15
	MASM	1.25	140	22.03	9.9	33.37
	ASM86	1.5	188	20.24	4.8	45.28
	PLS86	10.0	808.78	76.43	49.24	92.11

Figure 18. Difficulty ranges on each set of data

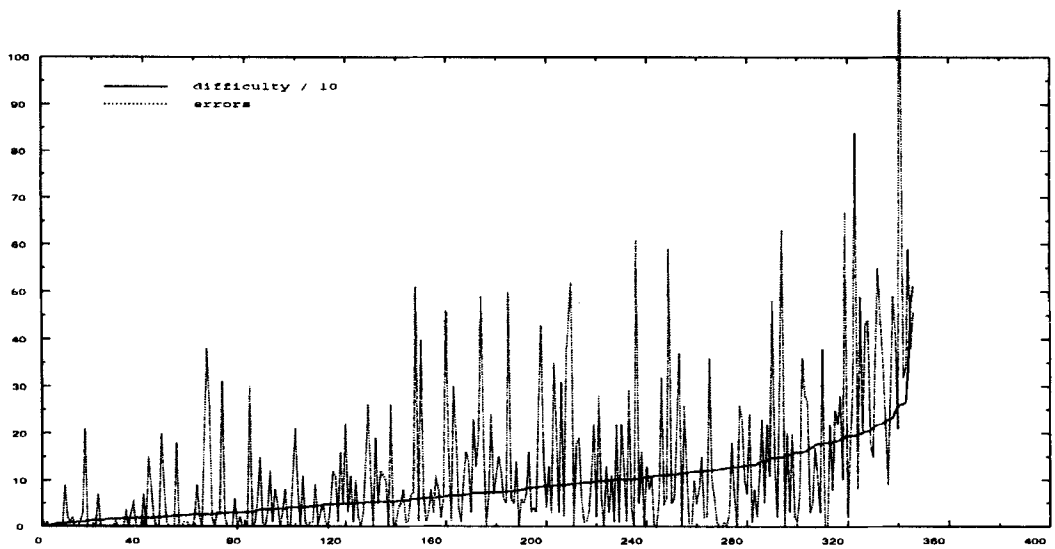


Figure 19. PU 4381: program difficulty and errors plotted against the module index (horizontal axis)

A commonly accepted proposition that high level languages are easier than assembler is partially supported by the data of Figure 18. The SPIEL and PL8 groups of modules have considerably lower average difficulty (44·51 and 24·37 respectively) than the assembler languages whose average difficulty ranges from 60 to 140. An exception occurs with the PLS/86 language which shows a difficulty of 76·43. Exceptions also occur with other SP 9370 sub-units such as MASM, ASM86, etc. but code in these sub-units may

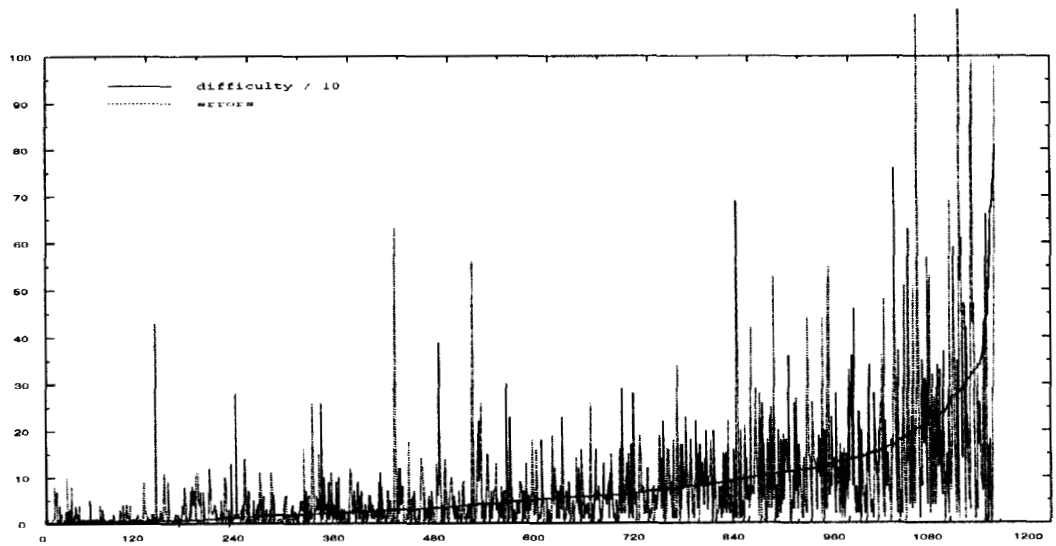


Figure 20. SP 9370 (PLS86): program difficulty and errors plotted against the module index (horizontal axis)

be considered negligible with respect to the code of the entire system. Smith (1980) reports average difficulty ranges from 91 to 561 for 370 assembly programs and average difficulty of 38 to 151 for PL/S programs. Compared with the findings of Figure 18 it seems that the microcode has less difficulty than 'regular' software. Figure 22 shows a typical plot between the program difficulty and the program size (in lines of code). The program difficulty appears to be highly correlated with the program size. The correlation coefficient between the two variables was found to be above 0·9 for most sub-units.

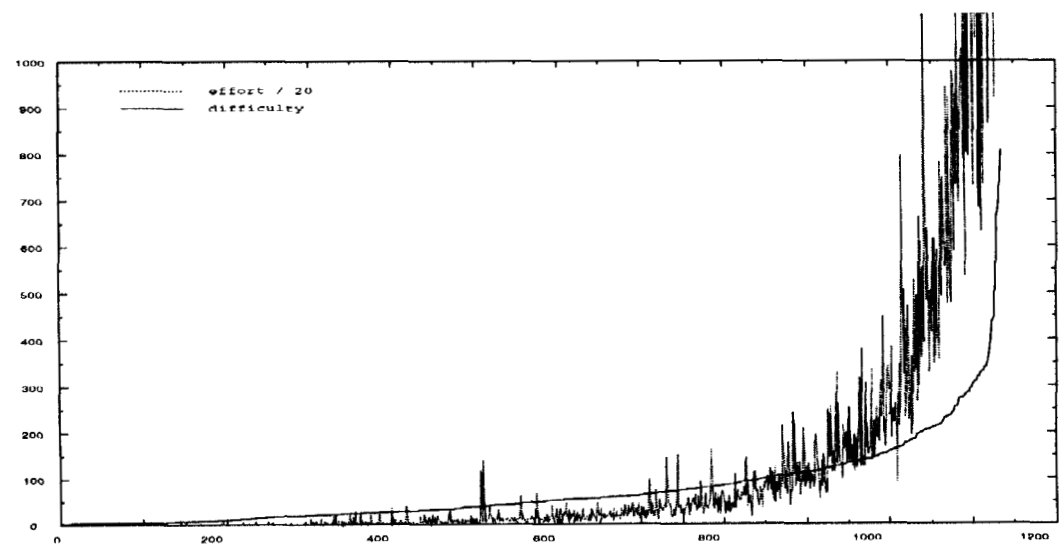


Figure 21. SP 9370 (PLS86): program difficulty and effort (horizontal axis = module index)

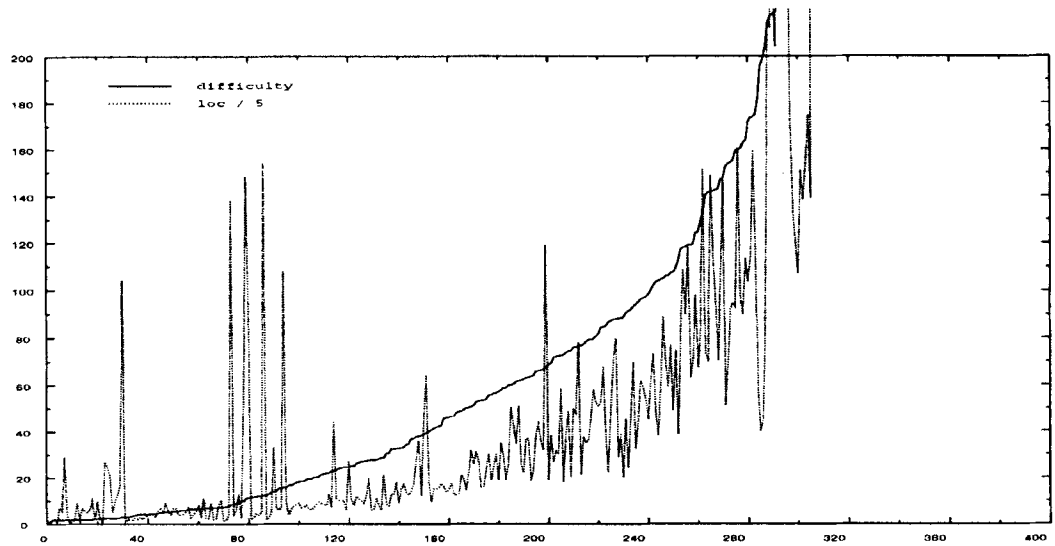


Figure 22. PU 9370: program difficulty and program size (horizontal axis = module index)

4.4. Language level

The language level is defined in software science as

$$\lambda = L^2 \times V = \frac{V}{D^2} \tag{2}$$

where, $L = 1/D$ is called the implementation level. The language level measures the programmers 'usage' of a particular language. Halstead postulated that the language level is constant for a given language independent of the algorithm being coded. The fact that two or more programs, written in the same language, have the same λ indicates that the programmers use the language in the same way. Halstead computed λ for a number of languages which indeed appeared to reflect Halstead's proposal. An analysis performed on 370 assembler and PLS/S code (Smith, 1980) has shown a relatively constant language level for the projects examined, as well. Figure 23 (taken from Smith (1980)) shows λ of eight languages. There are of course exceptions to λ being constant due to various reasons, such as code impurities, program structure, etc. In addition, in a large project there may be 'exceptional' modules, that do not fit well with the rest of the code. Smith

language	λ	st. dev.
ENGLISH	2.16	0.74
PL/S	2.05	1.14
PL/I	1.53	0.92
ALGOL 58	1.21	0.74
FORTRAN	1.14	0.81
PILOT	0.92	0.43
IBM 360/370 BAL	0.91	0.79
CDC 6500 BAL	0.88	0.42

Figure 23. Language level of English and several computer languages

(1980) identifies several such modules, that usually contain declaration statements, tables, message processing code, which tend to repeatedly use the same operators. This drives the language level very high.

The language level for our data is shown in Figure 24, along with the standard deviation. Significantly higher numbers were obtained in our data than those published by Smith and Halstead. In most cases the language level is uniform. In some cases, however, such as ASM2 and ASM3 sub-units of the SP 4381 system, extremely high standard deviation was obtained, which in some cases can be attributed to exceptional modules. In these cases, while most modules had 'regular' values, there were a few with two orders of magnitude difference. These modules caused the average λ and its standard deviation to be high. Elimination of these modules resulted in signficantly better results for these sub-units.

Manual examination of several modules with exceptionally high language level (several hundreds or thousands) showed that these modules contained data tables of message processing code which by nature result in high repetition of just a few operators. For example a module of the ASM3 SP 4381 unit was found to contain data tables consisting of 582 lines of code, 4 unique operators, 581 total operators, 1 234 unique operands and 1 286 total operands. This particular module resulted in volume $V = 19\,181{\cdot}17$ units, difficulty $D = 2{\cdot}08$ and language level $\lambda = 4\,433$. The elimination of this module from the average λ resulted in an improvement from 39·40 to 18·27.

The exceptional modules can be held responsible for only part of the variation of language level. Since only a few modules of our code contain tables/declarations that can drive λ very high, the large variation of λ in our data cannot support Halstead's claim that 'λ remains constant for any one language'. We conclude this section by noting that serious questions about the validity of this assertion have been raised in the past, i.e., Lister (1982) where it was demonstrated that if this assertion was true than all algorithms with the same number of input/output parameters could be written with the same effort.

5. ERROR PREDICTION BASED ON SOFTWARE METRICS

As indicated previously, the software science metrics have been used in a variety of applications, including software project planning, product reliability and error projection

4381 Computer System				9370 Computer System			
unit	Language	l. level	st. dev.	unit	Language	l. level	st. dev.
PU	ASM1	2.52	4.24	PU	ASM4	19.35	94.45
IO	ASM1	2.15	1.47		ASM5	11.14	59.06
	ASM2	6.22	35.47	IO	PL8	15.26	44.00
SP	SPIEL	3.75	4.82		680ASM	6.33	17.19
	ASM3	39.40	305.89		ASM4	1.34	1.62
					370 ASM	133.79	483.95
				SP	MASM	9.95	9.47
					ASM86	1200.13	2540.65
					PLS86	5.11	11.97

Figure 24. Language level for each sub-unit

(Conte, Dunsmore and Shen, 1986; Triantafyllos, Vassiliadis and Kobrosly, 1992). The premise for their wide use is that software science metrics can capture program characteristics which can be mathematically formulated and used in relevant studies. Several static error prediction models have been developed that use, in one way or another, software metrics. Examples of such models can be found in Conte, Dunsmore and Shen (1986) and Triantafyllos, Vassiliadis and Kobrosly (1992). Static models are of increasing interest, assuming their applicability be granted, because the errors expected in a project can be determined as soon as the code is written and the metrics counted.

One category of static error prediction models, called linear regression models, assume that some parameters, usually code metrics, can predict the errors in a module using a linear relationship between error and metrics (i.e., errors = $b_0 + b_1 \times \text{metric} + \ldots$). Using empirical data from a completed project, the coefficients of the above model may be estimated, so that the equation best fits the errors found in the project. Such a model can be a valuable source of information if it can be shown that the model may be successfully applied in future projects to estimate the errors expected to be found during their development. The above model may be expanded to include more metrics resulting in a multilinear regression model.

In this section we examine the possibility of using the metrics introduced previously to predict the errors found during the testing of microcode. In particular, we investigate whether each of the independent metrics, reported in Triantafyllos, Vassiliadis and Delgado-Frias (1993a), can be used in a linear model to predict the errors found during the testing of a microcode module. Each metric, examined in this section, is assumed to be a linear predictor of the errors having the following form:

$$\text{errors} = b_0 + b_1 \times \text{metric} \tag{3}$$

and

$$\text{errors} = b_0 + b_1 \times \text{metric1} + b_2 \times \text{metric2} + \ldots \tag{4}$$

We used linear regression on each set of our data to compute the coefficients of the above model, using the LOC metric, operands, operators, complexity, effort and function calls. In each case we compute the correlation coefficient ρ and the coefficient of determination R^2. The results of our investigations are reported in the sections to follow.

Figure 25 shows the correlation between errors and several metrics for each sub-unit of our data. As the figure suggests, none of the metrics examined is a good estimator of the errors found.

Additional investigation performed on each sub-unit showed that using the above metrics as error predictors resulted in less than 10% of the modules having predicted errors within $\pm 10\%$ of the actual errors. Figures 26 and 27 show typical plots of the actual versus predicted errors. An exception occurred in the 4381 IO unit which throughout this investigation produced better results.

Since several metrics were found to be linearly independent, a question arose whether two or more of these metrics can predict the errors of the system better than one metric. We used the regression on the following equations: the models assumed in this experiment are:

$$\text{errors} = b_0 + b_1 \times \text{LOC} + b_2 + \text{complexity} \tag{5}$$

unit	Language	LOC	Complex	Operator	Oprd	Length	Effort	f. Calls
IBM 4381 Computer System								
PU	ASM1	0.530	0.423	0.447	0.435	0.440	0.495	0.156
IO	ASM1	0.905	0.820	0.898	0.900	0.899	0.811	0.454
SP	ASM2	0.343	0.275	0.300	0.295	0.297	0.267	0.304
	SPIEL	0.432	0.426	0.413	0.435	0.434	0.392	0.408
	ASM3	0.351	0.264	0.334	0.332	0.333	0.261	0.301
IBM 9370 Computer System								
PU	ASM4	0.611	0.677	0.577	0.593	0.588	0.340	0.544
IO	ASM5	0.755	0.730	0.665	0.659	0.662	0.665	0.684
	PL8	0.377	0.539	0.337	0.225	0.287	0.208	0.421
	680ASM	0.918	0.893	0.910	0.913	0.912	0.829	0.016
SP	ASM4	0.277	0.144	0.324	0.273	0.292	0.198	0.176
	370 ASM	0.456	0.483	0.456	0.461	0.459	0.404	0.462
	MASM	0.722	0.739	0.715	0.723	0.720	0.757	0.665
	ASM86	0.208	-0.125	0.206	0.253	0.238	-0.032	-0.135
	PLS86	0.568	0.513	0.533	0.538	0.536	0.326	0.396

Figure 25. Correlation between errors and number of metrics

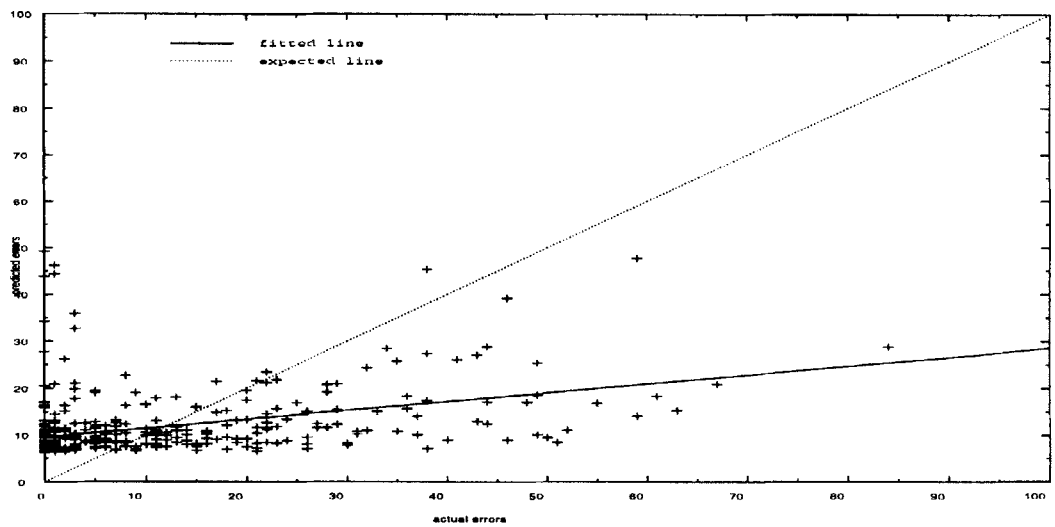


Figure 26. PU 4381: actual versus predicted errors based on operands

$$\text{errors} = b_0 + b_1 \times \text{LOC} + b_2 \times \text{function calls} \quad (6)$$

$$\text{errors} = b_0 + b_1 \times \text{LOC} + b_2 \times \text{complexity} + b_3 \times \text{effort} \quad (7)$$

$$\text{errors} = b_0 + b_1 \times \text{LOC} + b_2 \times \text{complexity} + b_3 \times \text{function calls} + b_4 \times \text{effort} \quad (8)$$

Figure 28 shows the coefficient of the multiple determination R^2, for each equation and

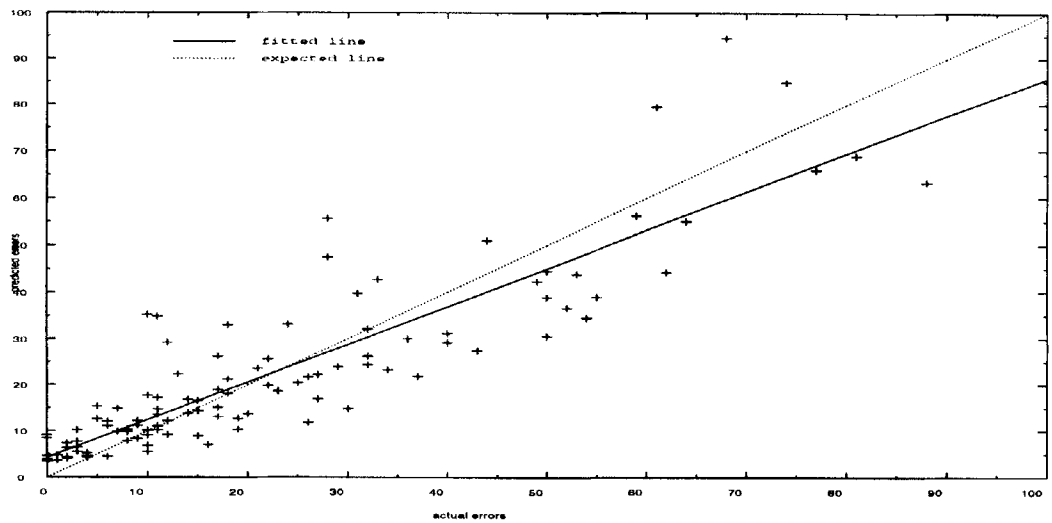


Figure 27. IO 4381: actual versus predicted errors based on operands

unit	Language	(5)	(6)	(7)	(8)
IBM 4381 Computer System					
PU	ASM1	0.280	0.285	0.294	0.306
IO	ASM1	0.824	0.831	0.826	0.836
SP	ASM2	0.117	0.119	0.121	0.126
	SPIEL	0.198	0.192	0.218	0.218
	ASM3	0.126	0.122	0.122	0.139
IBM 9370 Computer System					
PU	ASM4	0.468	0.419	0.516	0.521
IO	ASM5	0.629	0.571	0.635	0.652
	PL8	0.317	0.178	0.350	0.367
	680ASM	0.841	0.854	0.888	0.170
SP	ASM4	0.084	0.092	0.154	0.170
	ASSEMBLE	0.233	0.216	0.277	0.282
	MASM	0.233	0.535	0.545	0.401
	ASM86	0.039	0.061	0.067	0.0701
	PLS86	0.324	0.328	0.394	0.395

Figure 28. Correlation expressed as R^2 between errors estimated from equations (5) to (8) and counted errors

each sub-unit. Three criteria are used to decide whether or not a multilinear model is capable of predicting the errors in a microcode system.

1. The coefficient of multiple determination R^2 must be high enough to warrant that much of the variance of the errors can be explained by the variation in the metrics.
2. The percentage of files whose predicted errors are within ±10% (90% accuracy) must be high.
3. The coefficients of the model ($b_0, b_1, \ldots$) used must be very similar for all groups of modules where regression is applied. This criterion is very important in prediction using regression models because once a model has been found to be a good error predictor it will be applied to predict the errors of a system using the coefficients computed on the system from where the model was derived. If the coefficients are not similar during the experimentation, there is no warranty that the same coefficients will predict the errors of another project.

Figure 28 shows the coefficient of multiple determination R^2 for each estimating equation and errors. As the figure suggests the coefficient is low, indicating a weak relationship between errors and each of the metrics used in the model. Additional analysis has shown that only a few modules had predicted errors within ±10% of the actual errors. The coefficient of the regression line in each case varies considerably. The above reasons led us to conclude that none of the models examined can be used in error prediction.

6. CONCLUSIONS

In this paper we reported some of the findings of a case study involving the microcode of the IBM ES/4381 and the ES/9370 computer systems. In particular, we calculated several metrics, investigated the relationships between them, identified the metrics that are linearly independent, investigated the applicability of several software science aspects to microcode, and finally we used the metrics in regression models to determine whether they can be used as reliable error estimators.

Regarding the relationships among several software science metrics our findings can be summarized as follows:

- Most modules (72%) of the microcode are small or medium size (less than 400 lines of code). The number of modules with large number of lines decreases exponentially. A very small percentage of modules have more than 4000 lines of code. There is no indication that the size of the program is related to program errors.
- The estimated program length (N) is highly correlated to observed program length ($\bar{N}$). The correlation coefficient, ρ, between N, $\bar{N}$ is above 0·9 in most sub-units; two sub-units exhibit $\rho = 0{\cdot}776$, the smallest correlation observed. This correlation, however, does not imply that the length equation accurately predicts the length of a module. The observation made in the past that $\bar{N}$ over-predicts smaller modules and under-predicts larger ones is partially true in our data. Three out of the 14 sub-units have clearly shown this trend, while the rest of the sub-units did not show conclusive evidence.
- The program difficulty metric D was found to be uncorrelated to microcode program errors. Our data show the difficulty as defined by Halstead does not indicate the

difficulty in writing a program which could be assumed to be correlated to program errors. The difficulty values obtained for the assembler languages was considerably less than the difficulty observed previously (Smith, 1980). This finding appears to suggest that microcode assembler code is less 'difficult' than 'regular' assembler code. However, such a conclusion is hardly acceptable based on these numbers alone.

- Large discrepancies were found in the language level (λ) metric between units. This metric was postulated by Halstead to be relatively constant for a particular language, independent of the algorithm being coded. This was not evident in our code which produced a wide range of λ. Some of the variation of λ was removed by excluding the so called 'exceptional' modules but the final values were still far from similar values published in the past.
- There is strong linear relationship between lines of code, volume, length, operands and operators. More than 90% of the variation in the volume, length, operands and operators may be associated with variation in the LOC. This suggests that one of these metrics is sufficient in a linear regression model. Additional linearly dependant metrics, in a multilinear model, do not contribute additional information.
- The number of decisions in a program, denoted also as complexity, and the number of function calls did not correlate well with other metrics. This suggests that these two metrics may, in some cases, provide additional information, if they are used in a regression model.
- Unlike all other metrics examined, the program effort was found to increase exponentially with the LOC. A second degree model was used to approximate the effort using LOC. Furthermore, none of the metrics examined seemed to be able to provide a satisfactory explanation of the effort spent for the development of the code.
- Grouping of the modules into logical or functional units appears to be very important for applying statistical methods. It was found that even though two metrics possess a strong linear relationship in all units of a certain partition of the system, this relationship may not exist if another partition is chosen.

It was disappointing to discover that the lines of code, a simplistic metric that does not take into account any aspect of the program or algorithm, is as powerful an expression of the program size as other more sophisticated size metrics such as the operands, operators, length and volume. In fact, our analysis strongly suggests that there is no evidence that operands, operators, length or volume, are in any way superior to the lines of code. Our experiments with five assembler languages and three high level languages show that length and volume, even though in principle could be considered to be superior than the LOC, fail as program size measures across languages by not indicating advantages over LOC. There may be, however, instances in which the use of length or volume may be advantageous. For example, Gaffney (1981) has shown that in some cases it is possible to estimate the program length N, before coding begins. In this case, if N can be estimated more precisely than LOC can, then N may be advantageous over LOC.

In the investigation of whether software metrics can be used to predict the errors of the microcode, we used seven metrics in a linear model to establish a relationship between the metric and the expected number of errors to be found in a module. With the exception of the IO unit of the 4381 system, a weak relationship was found between the errors and each metric. None of the metrics examined produced acceptable results as a linear error predictor. Additionally, none of the metrics was found to be a better error predictor than

the LOC. Roughly speaking, our investigation suggests that the LOC metric is as good as any other metric when it comes to predicting the errors in a system using a linear model. This finding, which is a consequence of a previous finding (Triantafyllos, Vassiliadis and Delgado-Frias, 1993a) regarding the linearity between metrics, is rather disappointing because it is widely believed that complexity and length metrics capture more information about the difficulty of a program than the LOC, and therefore would provide more insight into the difficulty and consequently the bugs that are incurred during the development of a microcode program. Contrary to some evidence published over the past two decades regarding software products, our investigation suggests that software metrics cannot be used in a linear model to predict the errors of the microcode development.

Multilinear models, on the other hand did not prove to be successful either. Four models were used in this investigation, derived from combinations of linearly independent microcode metrics. The analysis has shown that multilinear models are not likely to be able to predict the errors in microcode systems. In most cases the coefficient of multiple determination was below 0·5. Using the models described in this section we computed the 'predicted errors' for each module. Less than 10 per cent of the modules tested had predicted errors within ±10 per cent of the actual errors. Based on our data, this case study concluded that software metrics used in multilinear models are not likely to be good error predictors for the microcode of computer systems.

References

Basili, V. R., Selby, R. W. and Phillips, T.-Y. (1983) 'Metric analysis and data validation across Fortran projects', *Transactions on Software Engineering*, **SE-9**(11), 652–663.

Christensen, K., Fitsos, G. P. and Smith, C. P. (1981) 'A prospective on software science', *IBM Systems Journal*, **20**(4), 372–387.

Conte, S. D., Dunsmore, H. E. and Shen, V. Y. (1986) *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Co., Inc., Reading, MA, 396 pp.

Gaffney, J. E. (1981) 'Software metrics: a key to improved software development management', in *Proceedings of the Computer Science Statistics 13th Annual Symposium on the Interface*, Carnegie-Mellon University, Pittsburgh, PA, pp. 211–220.

Halstead, M. H. (1977) *Elements of Software Science*, Elsevier North-Holland, Inc., New York, NY, 127 pp.

Henry, S. M. (1979) 'Information flow metrics for the evaluation of operating systems' structure', Doctoral dissertation, Computer Science Department, Iowa State University, Ames, IA, 145 pp.

Kafura, D. and Canning, J. (1985) 'A validation of software metrics using many metrics and two resources', in *Proceedings of the Eighth International Conference on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, pp. 378–385.

Lister, A. M. (1982) 'Software science—the emperor's new clothes?', *Australian Computer Journal*, **14**(1), 66–70.

McCabe, T. J. (1976) 'A complexity measure', *Transactions on Software Engineering*, **SE-2**(4), 308–320.

McClure, C. L. (1978) 'A model for program complexity analysis', in *Proceedings of the Third International Conference on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, pp. 149–157.

Motley, R. W. and Brooks, W. D. (1977) 'Statistical prediction of programming errors', Final Technical Report RADC-TR-77-175, Rome Air Development Center, Rome, NY, 39 pp.

Paulsen, L. R., Fitsos, G. P. and Shen, V. T. (1983) 'A metric for the identification of error-prone software modules', IBM Technical Report TR 03.228, IBM Santa Teresa Laboratory, San Jose, CA, 38 pp.

Shen, V. T., Conte, S. D. and Dunsmore, H. E. (1983) 'Software science revisited: a critical analysis of the theory and its empirical support', *Transactions on Software Engineering*, **SE-9**(3), 155–165.

Shen, V. T., Yu, T.-J., Thebaut, S. M. and Paulsen, L. R. (1985) 'Identifying error-prone software—an empirical study', *Transactions on Software Engineering,* **SE-11**(4), 317–324.

Smith, C. P. (1980) 'A software science analysis of IBM programming products', IBM Technical Report TR 03.081, IBM Santa Teresa Laboratory, San Jose, CA, 38 pp.

Triantafyllos, G., Vassiliadis, S. and Delgado-Frias, J. (1993a) 'An analysis of the microcode metric relationships', IBM Technical Report TR 01.C648, Endicott, NY, 117 pp.

Triantafyllos, G., Vassiliadis, S. and Delgado-Frias, J. (1993b) 'Trends in the microcode metrics', IBM Technical Report TR 01.C660, Endicott NY, 47 pp.

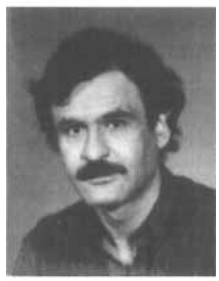
Triantafyllos, G., Vassiliadis, S. and Kobrosly, W. (1992) 'An evaluation of the accuracy of static error prediction models', IBM Technical Report TR 01.C579, Endicott NY, 120 pp.

Walston, C. E. and Felix, C. P. (1977) 'A method for programming measurement and estimation', *IBM Systems Journal,* **16**(1), 54–73.

Woodfield, S. (1980) 'Enhanced effort estimation by extending basic programming models to include modularity factors', Doctoral dissertation, Computer Science Department, Purdue University, West Lafayette, IN, 120 pp.

Authors' biographies:

George Triantafyllos worked at IBM's Laboratories at Endicott, NY and Poughkeepsie, NY from 1987 to 1995 on assignments on the automation of functional testing, the reliability of computer systems and the design of the Open System Adapter for the IBM 3090. At IBM, he received five awards, published more than thirty technical reports and papers, and filed three patent applications. Dr. Triantafyllos's research interests lie in computer architecture, hardware design, functional testing, software engineering, parallel processing and fuzzy logic. He started his electrical engineering education with a Diploma from the National Technical Institute of Athens, Greece in 1984. He also holds an electrical engineering degree from Fairleigh Dickinson University in Teaneck, NJ, and a masters in computer engineering from Syracuse University in Syracuse, NY. His doctorate is in electrical engineering from the State University of New York at Binghamton, NY in 1993.



Stamatis Vassiliadis worked for a decade with IBM in the Laboratories at Austin, TX, Poughkeepsie, NY and Endicott, NY. He worked as an engineer and chief architect on computer architecture and organization projects, and was involved in the design and implementation of the IBM 9370 model 60 computer. At IBM he received awards including 21 levels of the publication achievement award, 14 levels of the invention achievement award and the Outstanding Innovation Award for Engineering/Scientific Hardware Design in 1989. In 1990, he was awarded the highest number of patents in IBM. He is currently a professor at T U Delft and has previously held academic positions at Cornell University and SUNY at Binghamton. Dr. Vassiliadis has research interests in computer architecture, hardware design and functional testing, parallel processors, computer arithmetic, EDFI for hardware implementations, neural networks, fuzzy logic and systems, and software engineering. He holds a doctorate in electrical engineering from the Politecnico di Milano in Milan Italy and a doctorate in computer science from the University of Namur in Belgium.

José G. Delgado-Frias holds a Ph.D. from Texas A&M University. He has held academic positions at the University of Oxford and the National Autonomous University of Mexico. His research interests include parallel computers, neural networks and software development. He has co-authored over seventy technical papers and holds four patents in the USA. Currently he is an Associate Professor in the Department of Electrical Engineering at the State University of New York at Binghamton.